LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Java 8

basierend auf Folien von Florian Erhard

Java 8

Erschienen am 18. März 2014

- Verbessertes Contended Locking
- Projekt *Lambda*
- Erweiterungen der Collections-API (Streams)
- Neue Date and Time API
- Verbesserungen bei der Garbage Collection, Reflection, Collections, Generics und Annotations
- Weitere kleinere Sprachverbesserungen wie beispielsweise Annotationen für Java-Typen
- Security-Verbesserungen
- Performanceverbesserungen
- Default Methoden in Interfaces
- Projekt Nashorn
- Java FX 8.0

Lambdas

```
public interface Comparator<T> {

    int compare(T o1, T o2);

}
```

Möglichkeit 1: Schreibe Klasse, die Comparator implementiert:

```
import java.util.Comparator;

public class MyComparator implements Comparator<Gene> {

    public int compare(Gene o1, Gene o2) {
        return o1.getSymbol().compareTo(o2.getSymbol());
    }

}
```

Lambdas

```
public interface Comparator<T> {

    int compare(T o1, T o2);

}
```

Möglichkeit 2: Schreibe innere Klasse, die Comparator implementiert:

```
public class XYZ {

    ....

    private class MyComparator implements Comparator<Gene> {
        public int compare(Gene o1, Gene o2) {
            return o1.getSymbol().compareTo(o2.getSymbol());
        }
    }
}
```

Lambdas

```
public interface Comparator<T> {

    int compare(T o1, T o2);

}
```

Möglichkeit 3: Schreibe anonyme Klasse, die Comparator implementiert:

```
public class XYZ {

    public static void main(String[] args) {
        ...
        Collections.sort(genes,new Comparator<Gene>() {
            public int compare(Gene p1, Gene o2) {
                return o1.getSymbol().compareTo(o2.getSymbol());
            }
        });
    }
}
```

Lambdas

```
public interface Comparator<T> {

    int compare(T o1, T o2);

}
```

Möglichkeit 4: seit Java 8: Lambda-Ausdruck

```
public class XYZ {

    public static void main(String[] args) {
        ...
        Collections.sort(genes,
    (Gene p1, Gene o2) -> o1.getSymbol().compareTo(o2.getSymbol())
        );
    }
}
```

Lambdas

```
public interface Comparator<T> {

    int compare(T o1, T o2);

}
```

Möglichkeit 5: seit Java 8: Lambda-Ausdruck; Typen kann man meist weglassen

```
public class XYZ {

    public static void main(String[] args) {
        ...
        Collections.sort(genes,
    (o1, o2) -> o1.getSymbol().compareTo(o2.getSymbol())
        );
    }
}
```

Lambdas

```
public interface Comparator<T> {

    int compare(T o1, T o2);

}
```

Möglichkeit 6: seit Java 8: Methoden-Referenz

```
public class XYZ {

    private static int compareSymbols(Gene o1, Gene o2) {
        return o1.getSymbol().compareTo(o2.getSymbol());
    }

    public static void main(String[] args) {
        ...
        Collections.sort(genes, XYZ::compareSymbols);
    }
}
```

Lambdas

- Interfaces mit genau einer Methode heißen *funktionale Interfaces*.
- Ein passender Lambda-Ausdruck kann überall dorthin geschrieben werden, wo ein Objekt erwartet wird, das ein funktionales Interface implementiert
- Passend ist der Lambda-Ausdruck dann, wenn seine (inferierten) Parametertypen und sein Rückgabetyp mit denen der Methode des funktionalen Interfaces übereinstimmt
- Lambdas sind sog. *Closures*, erben also die Variablen des definierenden Kontexts
- in Java müssen diese allerdings „effectively final“ sein

- Eine Methodenreferenz ist eine Kurzschreibweise für einen Lambda-Ausdruck:

```
XYZ::compareSymbols
```

```
(o1,o2)->XYZ.compareSymbols(o1,o2)
```

- Methodenreferenzen gibt es für statische Methoden (Klasse::methode), für Instanzvariablen (Objekt::methode) und Konstruktoren (Klasse:new)
- Oft hilfreich: Referenzen auf API-Methoden: z.B. Math::max

Lambdas

Wichtige funktionale Interfaces aus der Java API (java.util.function.*)

Interface	Typ
Predicate<T>	(T)->**boolean**
Function<T,R>	(T)->R
BiFunction<T,U,R>	(T,U)->R
Consumer<T>	(T)->**void**
Supplier<T>	()->T
UnaryOperator<T>	(T)->T
BinaryOperator<T>	(T,T)-T

Sowie angepasste für einige primitive Datentypen (z.B. IntToDoubleFunction)

Streams

Umgang mit Collections bisher:

```
for (Gene g : genes) {
    if (g.getBiotype().equals(type)) {
        System.out.println(g.getSymbol());
    }
}
```

Jetzt mit Streams:

```
genes.stream()
        .filter(g->g.getBiotype().equals(type))
        .map(s->s.getSymbol())
        .forEach(System.out::println)
```

jede Collection kann stream erzeugen

Filtern mit Predicate

Mappe mit Function

Gib alles ankommende aus

Vorteil: kürzer, konziser, **parallelisierbar**
Nicht nur bei Collections: Arrays.stream(T[] a), BufferedReader.lines()

Streams

Unterscheide zwei Arten von Operationen auf Streams:

- „intermediate“: Produzieren wieder einen neuen Stream
- „terminal“: Beenden den Stream (Seiteneffekt oder Endergebnis)

```
genes.stream()
        .filter(g->g.getBiotype().equals(type))
        .map(s->s.getSymbol())
        .forEach(System.out::println)
```

filter und map sind intermediate Operationen, forEach ist terminal (ohne Ergebnis)

Weitere wichtige **intermediate** Operationen:
sorted,distinct,limit,peek,parallel,sequential

Weitere wichtige **terminale** Operationen:
sum, min,max,reduce,count,forEachOrdered,collect,allMatch

Fragen ?